**Week 1  
Module 1 – Design patterns and principles**

**Exercise 1 - Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Approach:**

1. Ensured only one instance of the Logger class exists globally for consistent logging.
2. Created a Java project SingletonPatternExample with a singleton package in Eclipse.
3. Used a private static instance, private constructor, and a public getInstance() method with lazy initialization.
4. Added a log(String message) method to simulate logging functionality.
5. Verified Singleton behaviour in the Main class using multiple references and console output.

**Code:**

1. **Logger.java:**

package singletonPattern;

public class Logger {

private static Logger *instance*;

private Logger()

{

System.***out***.println("Logger instance created.");

}

public static Logger getInstance()

{

if (*instance* == null)

{

*instance* = new Logger();

}

return *instance*;

}

public void log(String message)

{

System.***out***.println("Log: " + message);

}

}

1. **Main.java:**

package singletonPattern;

public class Main

{

public static void main(String[] args)

{

Logger logger1 = Logger.*getInstance*();

logger1.log("Starting application");

Logger logger2 = Logger.*getInstance*();

logger2.log("Performing an operation");

if (logger1 == logger2)

{

System.***out***.println("Both logger1 and logger2 refer to the same instance.");

}

else

{

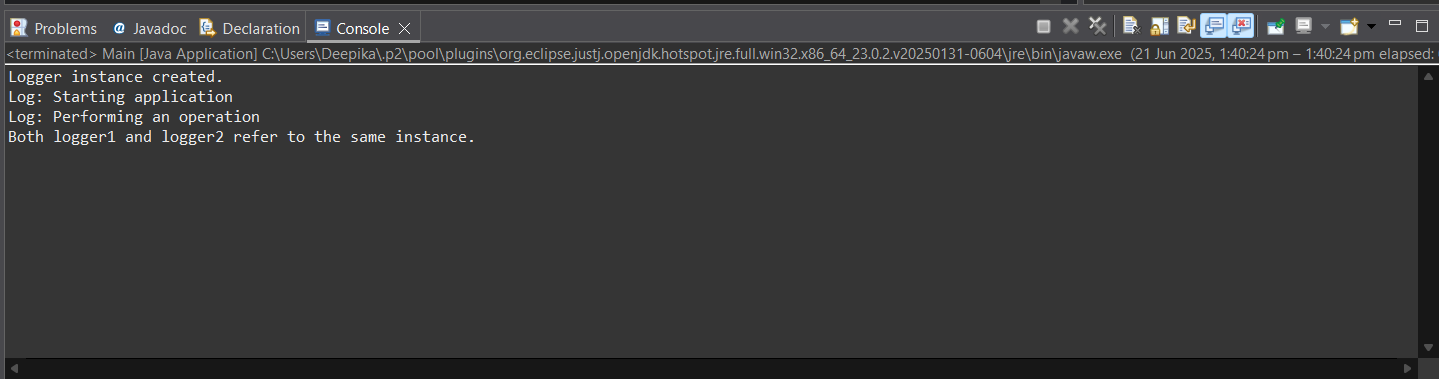
System.***out***.println("Different logger instances found (something's wrong).");

}

}

}

**Output:**

****

**Exercise 2 - Factory Method Pattern:**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Approach:**

1. **Designed a Common Interface:** Created a Document (or IDocument) interface with methods like open() for shared document behavior.
2. **Implemented Concrete Classes:** Built specific classes like WordDocument, PdfDocument, and ExcelDocument that implemented the interface.
3. **Created Factory Hierarchy:** Developed an abstract class DocumentFactory with a method createDocument(), and extended it with WordFactory, PdfFactory, etc.
4. **Tested with a Main Class:** Used the factory classes to create and open different document types, verifying the flexibility and reusability of the pattern.

**Code:**

1. **IDocument.java**

package factoryMethodPattern;

public interface IDocument {

void open();

String getDocumentType();

}

1. **WordFile.java**

package factoryMethodPattern;

public class WordFile implements IDocument {

*@Override*

public void open() {

System.***out***.println(">> Word document is now open in MS Word.");

}

*@Override*

public String getDocumentType() {

return "Word File";

}

}

1. **PdfFile.java**

package factoryMethodPattern;

public class pdfFile implements IDocument {

*@Override*

public void open() {

System.***out***.println(">> PDF document opened using Adobe Reader.");

}

*@Override*

public String getDocumentType() {

return "PDF File";

}

}

1. **ExcelFile.java**

package factoryMethodPattern;

public class ExcelFile implements IDocument {

*@Override*

public void open() {

System.***out***.println(">> Excel spreadsheet is now open in Excel.");

}

*@Override*

public String getDocumentType() {

return "Excel File";

}

}

1. **DocumentCreator.java**

package factoryMethodPattern;

public abstract class DocumentCreator {

public abstract IDocument generateDocument();

}

1. **WordDocCreator.java**

package factoryMethodPattern;

public class WordDocCreator extends DocumentCreator {

*@Override*

public IDocument generateDocument() {

return new WordFile();

}

}

1. **PdfDocCreator.java**

package factoryMethodPattern;

public class PdfDocCreator extends DocumentCreator {

*@Override*

public IDocument generateDocument() {

return new pdfFile();

}

}

1. **ExcelDocCreator.java**

package factoryMethodPattern;

public class ExcelDocCreator extends DocumentCreator {

*@Override*

public IDocument generateDocument() {

return new ExcelFile();

}

}

1. **Main.java**

package factoryMethodPattern;

public class Main {

public static void main(String[] args) {

DocumentCreator wordCreator = new WordDocCreator();

IDocument word = wordCreator.generateDocument();

System.***out***.println("Created: " + word.getDocumentType());

word.open();

DocumentCreator pdfCreator = new PdfDocCreator();

IDocument pdf = pdfCreator.generateDocument();

System.***out***.println("Created: " + pdf.getDocumentType());

pdf.open();

DocumentCreator excelCreator = new ExcelDocCreator();

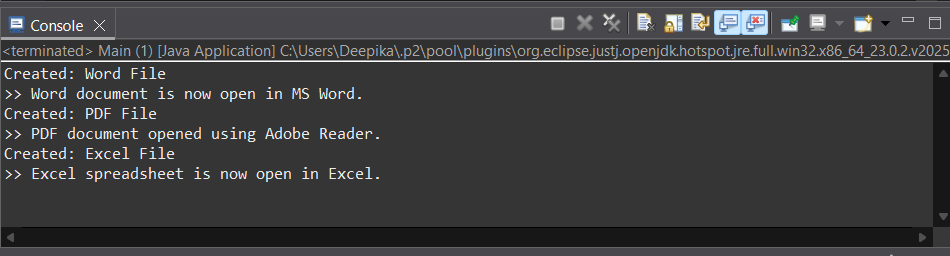
IDocument excel = excelCreator.generateDocument();

System.***out***.println("Created: " + excel.getDocumentType());

excel.open();

}

}

**Output:**

**Module 2 – Algorithms\_Data Structures**

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Approach:**

1. **Defined Product Structure:** Created a Product class with attributes like productId, productName, and category for realistic search scenarios.
2. **Implemented Search Methods:** Built both **linear search** (works on unsorted data) and **binary search** (requires sorted data) functions.
3. **Tested and Compared Performance:** Stored sample products in arrays, tested both search methods, and analysed which performs better for large datasets.
4. **Understand Asymptotic Notation:**

* **Big O Notation** describes how the runtime of an algorithm grows relative to input size (**n**).
* **Linear Search:** Best Case is O(1) (item at beginning) and Average/Worst Case is O(n)
* **Binary Search:** Best Case is O(1) and Average/Worst Case is O(log n). It requires sorted data.

1. **Code:**

**Product Class:**

package eCommercePlatform;

public class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

*@Override*

public String toString()

{

return "[" + productId + " | " + productName + " | " + category + "]";

}

}

**Search Functions:**

package eCommercePlatform;

public class ProductSearch {

public static Product linearSearch(Product[] products, String name) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

public static Product binarySearch(Product[] products, String name) {

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(name);

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

}

**Main Class:**

package eCommercePlatform;

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(101, "Shoes", "Footwear"),

new Product(102, "Laptop", "Electronics"),

new Product(103, "Shirt", "Apparel"),

new Product(104, "Watch", "Accessories"),

new Product(105, "Phone", "Electronics")

};

String searchName = "Watch";

Product result1 = ProductSearch.*linearSearch*(products, searchName);

System.***out***.println("Linear Search Result: " + result1);

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName));

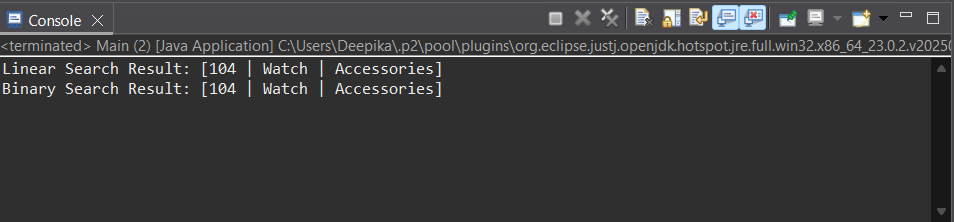
Product result2 = ProductSearch.*binarySearch*(products, searchName);

System.***out***.println("Binary Search Result: " + result2);

}

}

1. **Output:**

****

1. **Analysis:**

* **Linear Search** has a time complexity of **O(n)** and is suitable for **small datasets** or when the array is **unsorted.**
* **Binary Search** has a time complexity of **O(log n)** and is ideal for **large datasets** that are **sorted,** offering much better performance.
* Binary Search is more suitable for an e-commerce platform due to its faster performance on large, sorted datasets. Since product listings can be sorted by name or ID, binary search enables quick lookups. In contrast, linear search is less efficient as data grows. For scalability & instant search results, binary search is better choice.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Approach:**

1. **Applied Recursive Thinking:** Recognized that future value can be expressed recursively: FV(n) = FV(n-1) × (1 + growthRate).
2. **Implemented Recursive Method:** Created a method that recursively calculates the future value by reducing the year count in each call.
3. **Tested with Sample Data:** Used example values (e.g., ₹10,000 with 8% growth over 5 years) to verify correctness.
4. **Analysed Efficiency:** Identified time complexity as **O(n)**.
5. Noted that the recursion is safe here but can be optimized or converted to iterative for large inputs.
6. **Recursive Algorithms:**

* **Recursion** is a programming technique where a method calls itself to solve a smaller instance of the same problem.
* It simplifies problems like computing future values over a period by breaking them into sub-problems (e.g., calculate value for year n using value for year n-1).

1. **Code:**

package forecasting;

public class FinancialForecast {

// Recursive method to calculate future value

public static double calculateFutureValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

// Recursive call: FV = PV \* (1 + r)^n

return *calculateFutureValue*(currentValue, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double initialInvestment = 10000.0;

double annualGrowthRate = 0.08; // 8%

int forecastYears = 5;

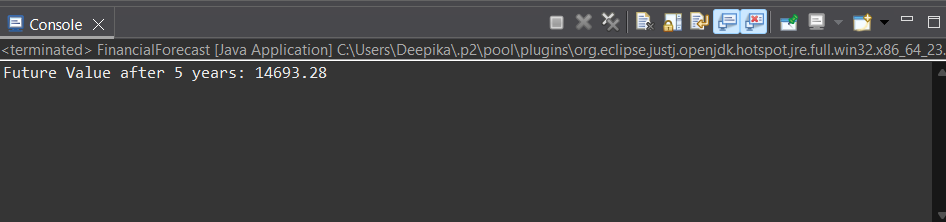
double futureValue = *calculateFutureValue*(initialInvestment, annualGrowthRate, forecastYears);

System.***out***.printf("Future Value after %d years: %.2f%n", forecastYears, futureValue);

}

}

1. **Output:**

****

1. **Analysis:**

* Time Complexity:
  + The time complexity of the above recursive method is **O(n),** where n is the number of years. Each recursive call reduces years by 1 until it reaches 0.
* Optimization Tip:
  + For simple linear recursion like this, performance is acceptable.
  + However, if results are reused across branches (e.g., in Fibonacci), use **memoization** or convert to an **iterative** solution to avoid recomputation and **stack overflow** risks for large n.